JRC 90-06

A NATURAL LANGUAGE INTERFACE TO DATABASES

Prepared by:

D.R. Ford
Johnson Research Center
The University of Alabama in Huntsville
Huntsville, AL 35899

Prepared for:

Tim Crumbley
System Software Branch
Information and Electronic Systems Lab
George C. Marshall Space Flight Center
Natinal Aeronautics and Space Administration
Marshall Space Flight Center, AL 35812

February 1990

TABLE OF CONTENTS

ABSTRACT

This paper presents the development of a Natural Language Interface (NLI) which is semantic-based and uses Conceptual Dependency representation. The system was developed using Lisp and currently runs on a Symbolics Lisp machine.

1.0 Natural Language Interface to Databases

Natural languages are the languages used by people in the course of their daily affairs, for example, English, French, Japanese, etc. Natural languages are used to express a broad range of ideas to others. Given enough attention, nearly any concept that comes to mind can be conveyed to another person through a common natural language. Some concepts are easy to express, such as, "I am hungry," whereas others may require lengthy explanations. The prime characteristic of natural languages is that they can be used to express nearly all the concepts that occur to the people who speak and understand them.

The word *natural* emphasizes a contrast with artificial languages. Artificial languages are those that have been designed to be highly expressive over a limited range of ideas. Musical notation is an artificial language. Another set of artificial languages is programming languages. These are interesting because, like natural languages, they can be used to express a broad range of concepts. LISP, for instance, is an extendable language, that is, if an idea is difficult to express in its current form, it can be improved at will. But programming languages have been designed with their application to computers in mind, and this has affected their form. Programming languages have been written so as to be analyzed easily by computers.

Research in natural language understanding is concerned with making computers capable of using natural languages. There are two reasons for this. First, computers that can use natural languages would undeniably be a useful tool. It would mean that a person in need of information retrieval or information processing on a computer could obtain it without having to learn a computer language or go through an intermediary. They would not have to worry about becoming fluent in a "foreign" language and maintaining that fluency just to

accomplish their jobs. A computer that could use natural languages could read normal text, providing users with access to computer-generated summaries or reports synthesized from reading several text sources.

The second motivation for natural language research is that it will increase our understanding of how human languages and minds work. To develop the technology for a computer to use language, we must first be able to say specifically what language is. We must be able to say precisely how the concepts we wish to express can be represented in the computer. Building computer programs requires this precision and attention to detail. A programming implementation of a theory of language can be used to identify flaws, inconsistencies, and areas of incompleteness that may go unnoticed.

1.1 Task Statement

The purpose of this task was to develop an interface to a database in order to determine the feasibility of such an interface. Also, the desirability of the interface was to be addressed. In addition, a secondary task was to gain a better understanding of what the capabilities of such an interface should be, and to determine some of the limits of this type of interface.

1.2 Task Conditions

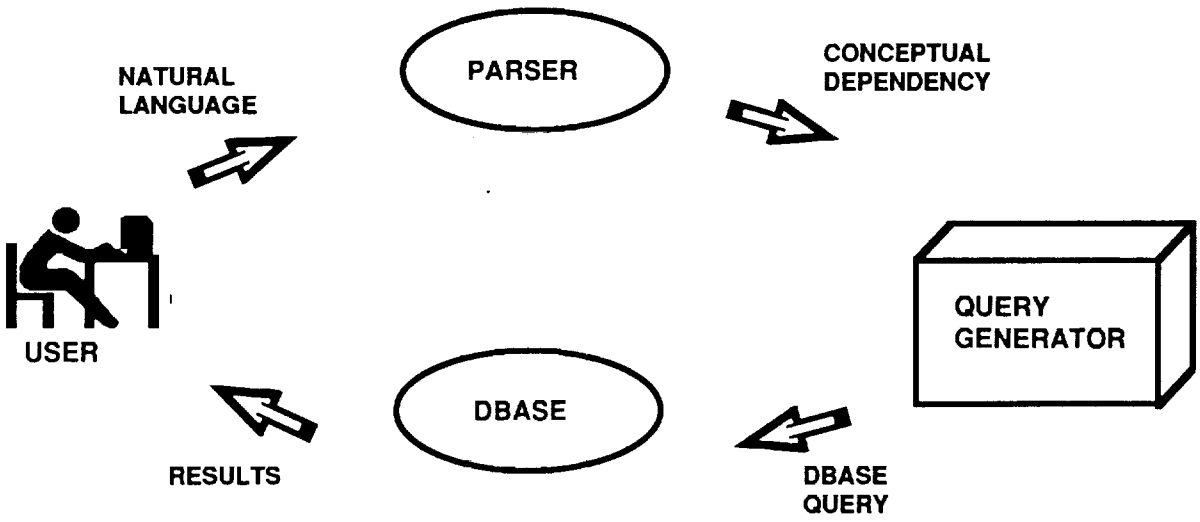
The natural language interface was to be developed on a Symbolics 3600 series Lisp machine using Symbolics Lisp. The interface should understand a limited subset of English in order to allow a user to query a database. The database should be located on another computer other than the Symbolic's machine that contained the interface. In addition, it should be on a completely different computer from the Symbolic's. There were no assumptions

or conditions placed on the means of communication between the Symbolic's machine and the computer with the database.

1.3 Task Approach

A NL Query generator prototype is being developed at the Johnson Research Center at the University of Alabama in Huntsville wherein queries, in natural language are generated for dbaseIII+. The main program resides on a symbolics 3620 machine while the database resides on an IBM personal computer. The database is manipulated through commands from the Symbolics. The communication protocol is established via RS 232. The process of database query and results are shown by the following figure:

A Graphic representation of the Natural Language Query Generator



The user types in a query in natural language on the symbolics. The parser translates it into Conceptual Dependency representation and generates a dbase query which is communicated to the PC via the RS 232. The RS 232 was chosen over others as the main idea was to set up communications between the Symbolics and the P.C. The manipulation is performed on the database and the results are communicated back to the user on the symbolics. The significant point of the exercise is that the user is not restricted to using specific dbase commands for manipulating the database. He can do so in the manner and language he prefers (provided it is in English).

The database is a simple one designed to represent the student records. It has been designed more to test the execution of the program and the generated queries. The following is a section of the database:

RECORD #	NAME	ST. NUM	S. S. NUM	SEX	AGE
1	Ash	12345	[redacted]	m	26
2	Dion	67897		m	28
3	John	70601		m	27
4	Mike	46893		m	26
5	Lisa	25789		f	24
6	Eddie	71214		m	24
7	Linda	45109		f	35
8	Cynthia	28633		f	25
9	Ben	65432		m	23
10	Gedro	64646		m	33
11	Paige	29099		f	24
12	Bernie	10001		m	38
13	Donnie	53200		m	37
14	Darlene	66677		f	25
15	Rica	99999		f	20

Execution of Database Manager Program on the P.C

The DB_MGR.PRG program first runs the basic program GETQUERY.BAS which receives the query from the Symbolics and writes a Dbase III program called QUERY.PRG. DB_MGR.PRG then executes QUERY.PRG storing the results in RESULTS.TXT. DB_MGR.PRG then runs the basic program SENDRES.BAS which sends the contents of RESULTS.TXT back to the Symbolics. Finally, DB_MGR.PRG loops back to GETQUERY.BAS, waiting for the next query from the Symbolics.

The words and the expressions are all defined in the dictionary. The database can handle all display and retrieve DBase commands from the Natural Language Query generator.

Working Examples of the Interface

1. The first example demonstrates the use of the verb "List" The word " List" is a dbase III command which performs , as the name suggests, the function of Listing the field names, required by the user. In this case, the user wants the Natural Language Query Generator (NLQG) to generate a query for listing all the males in the database. The user presses the Select ■ key and the Natural Language Query Generator is displayed on the screen, with the prompt- **Query.** The user then types in the command: **" List all males".** The NLQG generates the query in Conceptual Dependency, the communication protocol is established with the P.C. and the command chain is established in the manner described above. The command is then executed in dbase III and the results flash for a second on the P.C. before they are communicated to the Symbolics and displayed there. The user types in his/her query in the the top half of the screen and the results are displayed on the bottom half of the screen.

The response to the query " List all males" is the Record number and the Names of all the males in the database. A printout of the screen is displayed below:

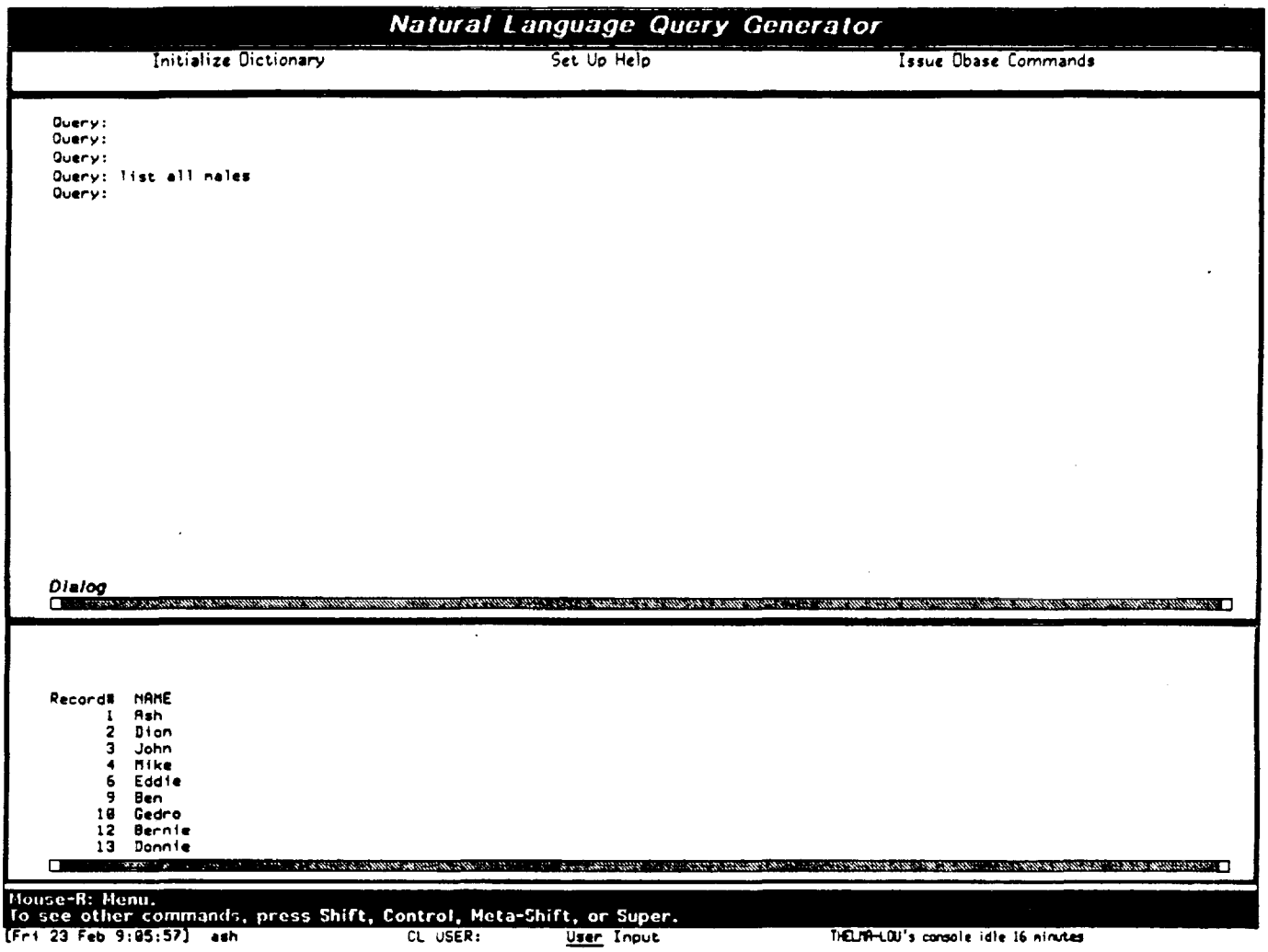


Figure 1

2. In the manner stated above, if the user wants to retrieve the names of all the females in the data base, he/she types in the command " List all females " The names of all the females in the data base will be displayed in the bottom half of the screen. The text above the output, i.e.

(DISPLAY DB-FIELD-VALUE (DB-FIELD-VALUE VALUE ("f") OPERATOR ("=")

FIELD (SEX)))

is the Conceptual Dependency representation of the typed in text. It tells the computer to look in the data base records which have a value of " f " (meaning female) in the field "Sex" and display the contents of all the fields in the records matching the search. The query and the output are shown in the following figure:

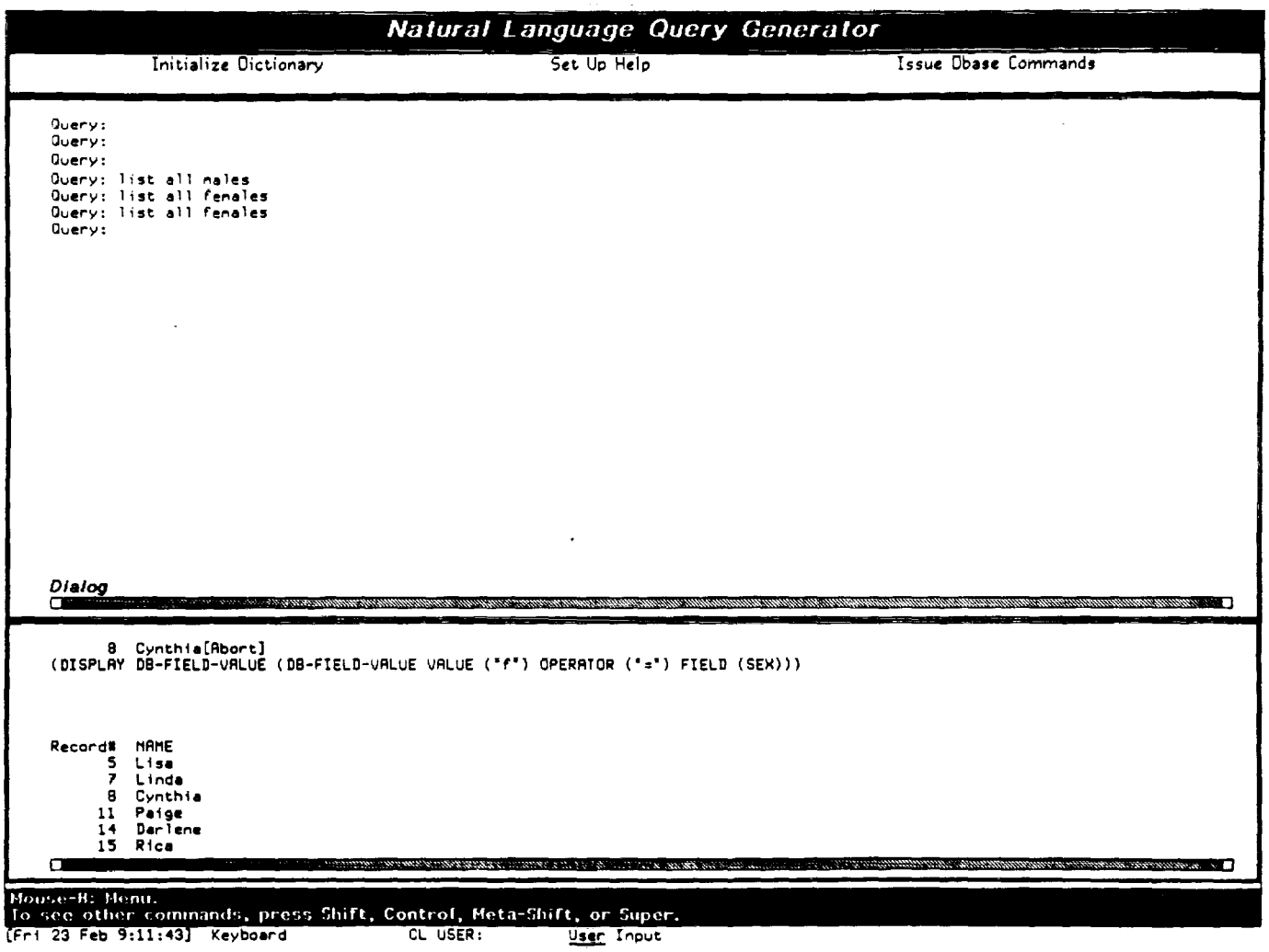


Figure 2

3. In this example, the user wants to retrieve the name of women, who are more than 30 years of age. However, he is not restricted to the "List" or "Display" commands, which are DBase commands. He/she can just type in "show" and whatever records he/she wants to get and the Query Generator will retrieve it for him/her. Only one record matches the query and the same is displayed at the bottom of the screen. The output is shown in the following figure:

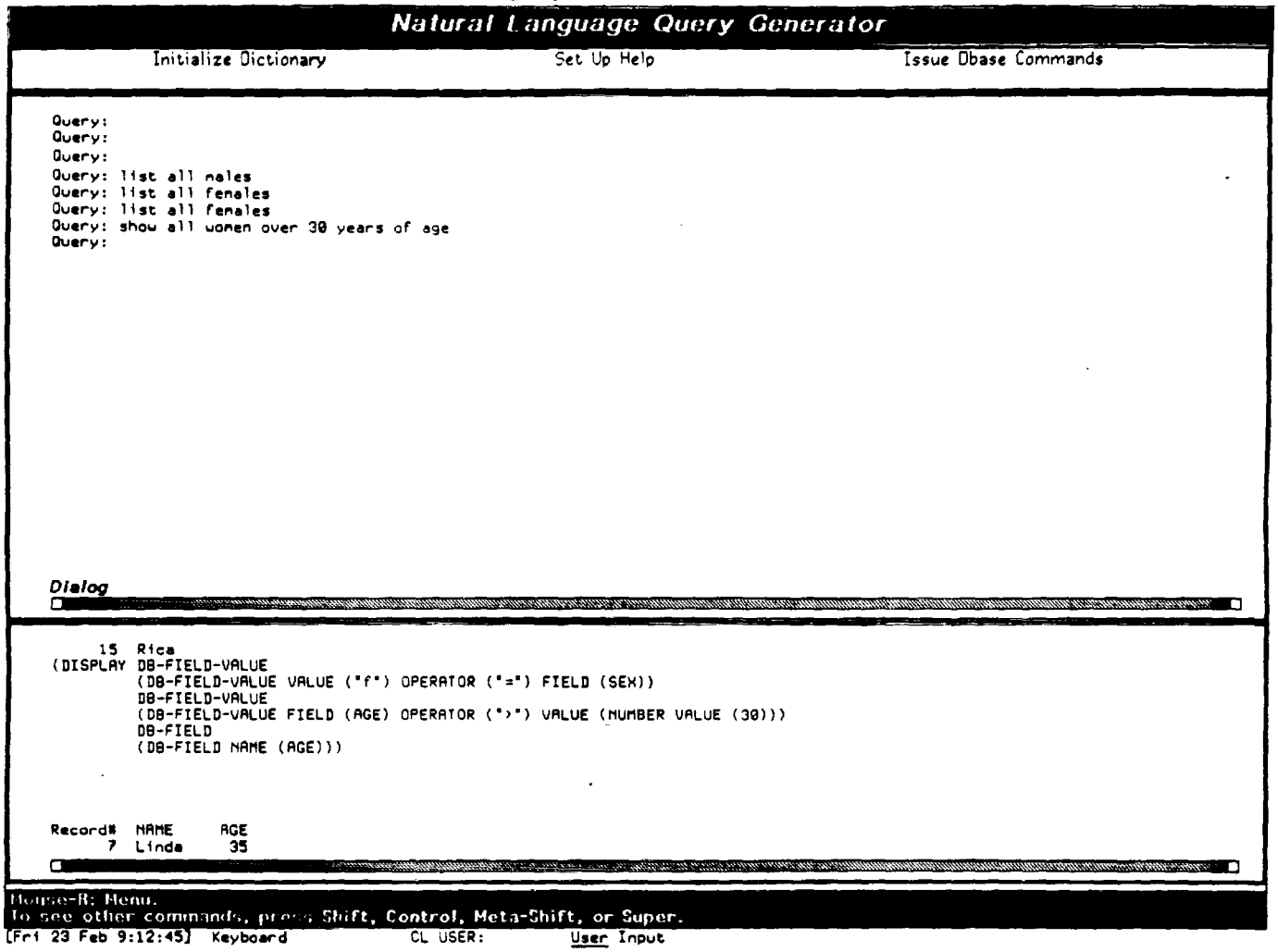


Figure 3

4. To demonstrate the capability of the NLQG, we can use the following example. The user can use any word synonomous with "List " or "Display" in the manner and the

NLQG will retrieve the records required. The output and the query are shown in the following figure:

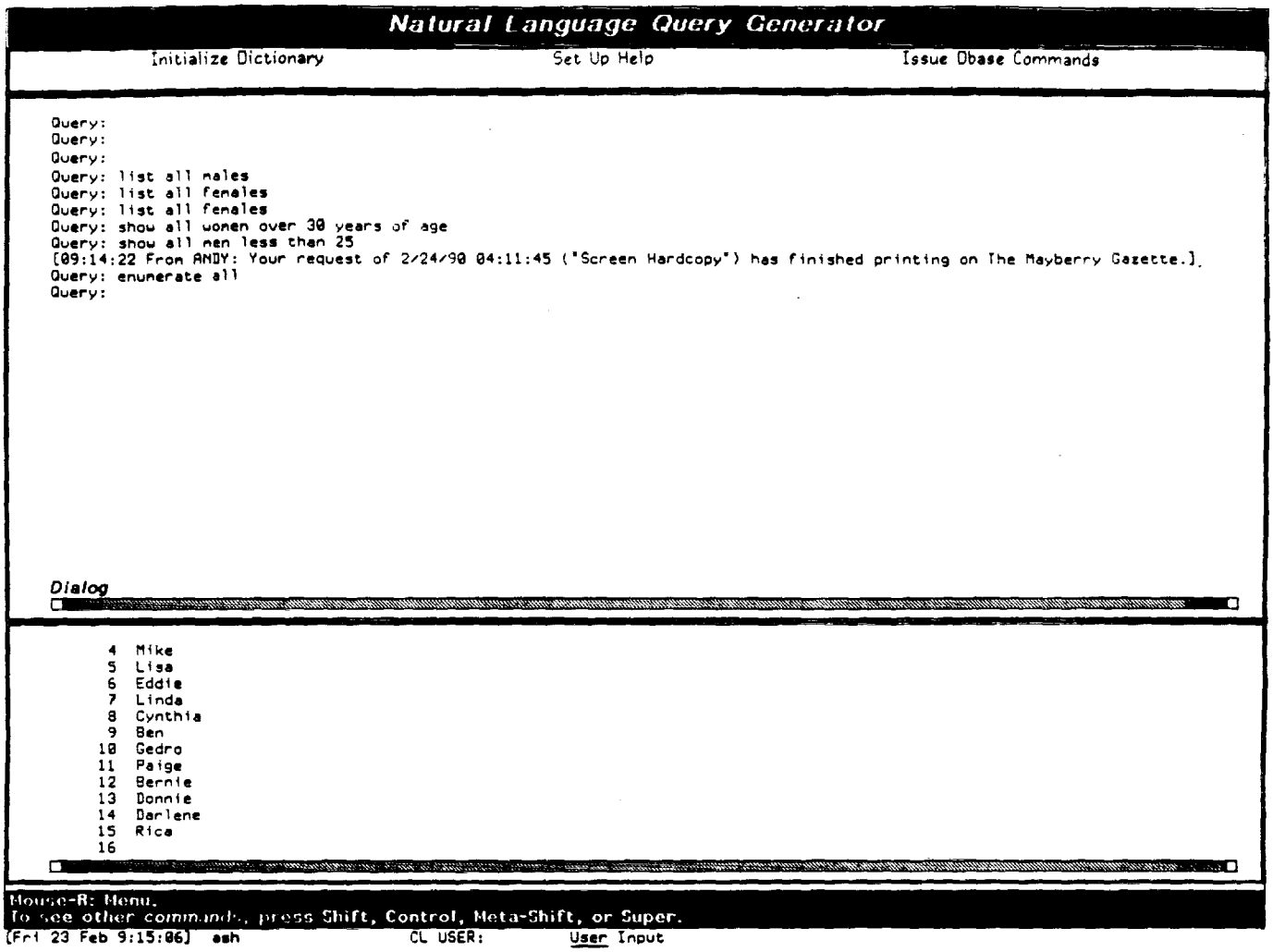


Figure 4

5. In this example, the user wants to retrieve the name and Social Security Number of all the males in the data base. In stead of using the word "Retrieve," he/she uses the word "Get." The query and the result are shown in the folllowing figure:

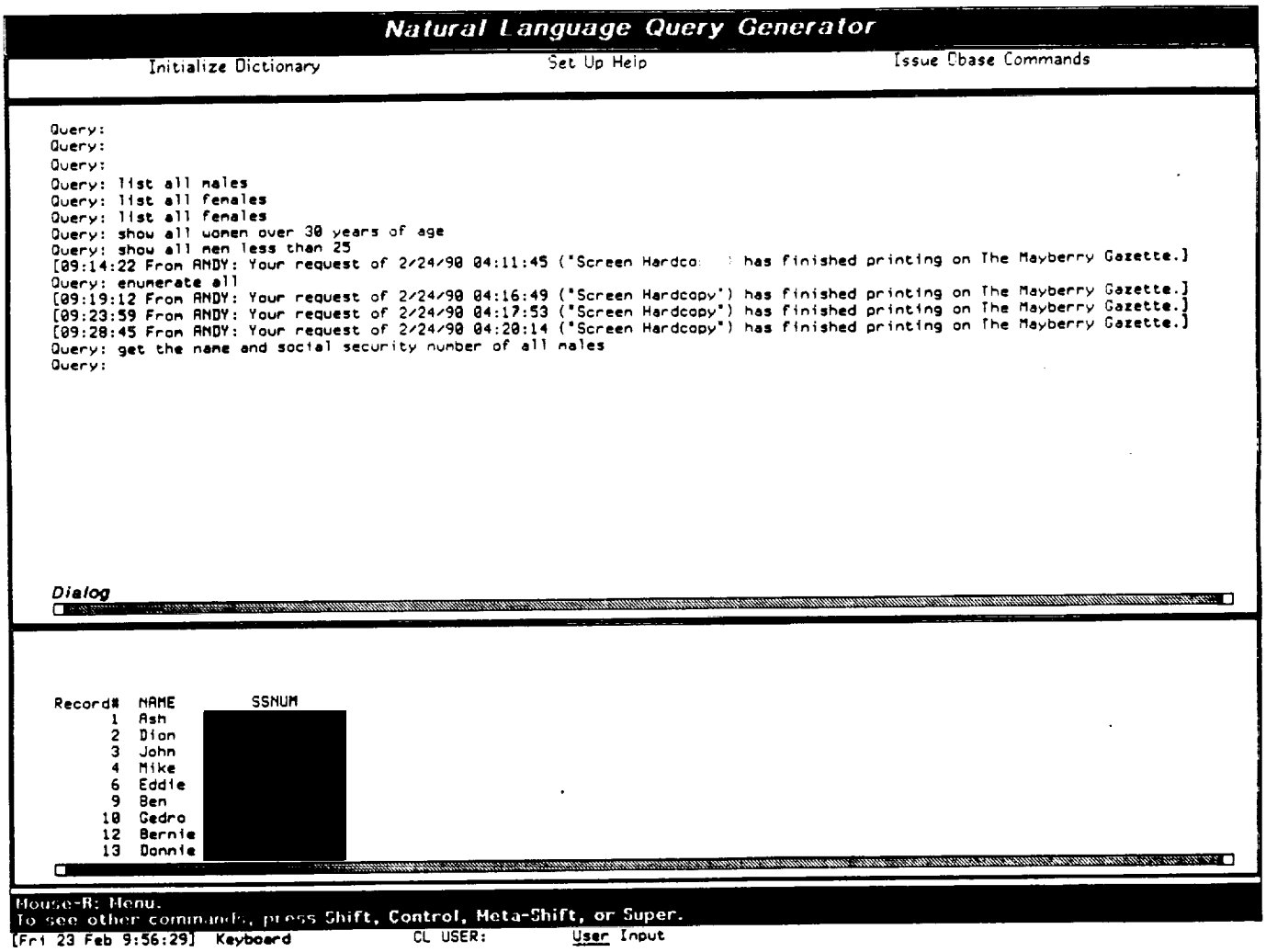


Figure 5

6. In this example, the user uses the word "Retrieve" to ddisplay the name and student number of all the males in the data base. The query and the output are shown below:

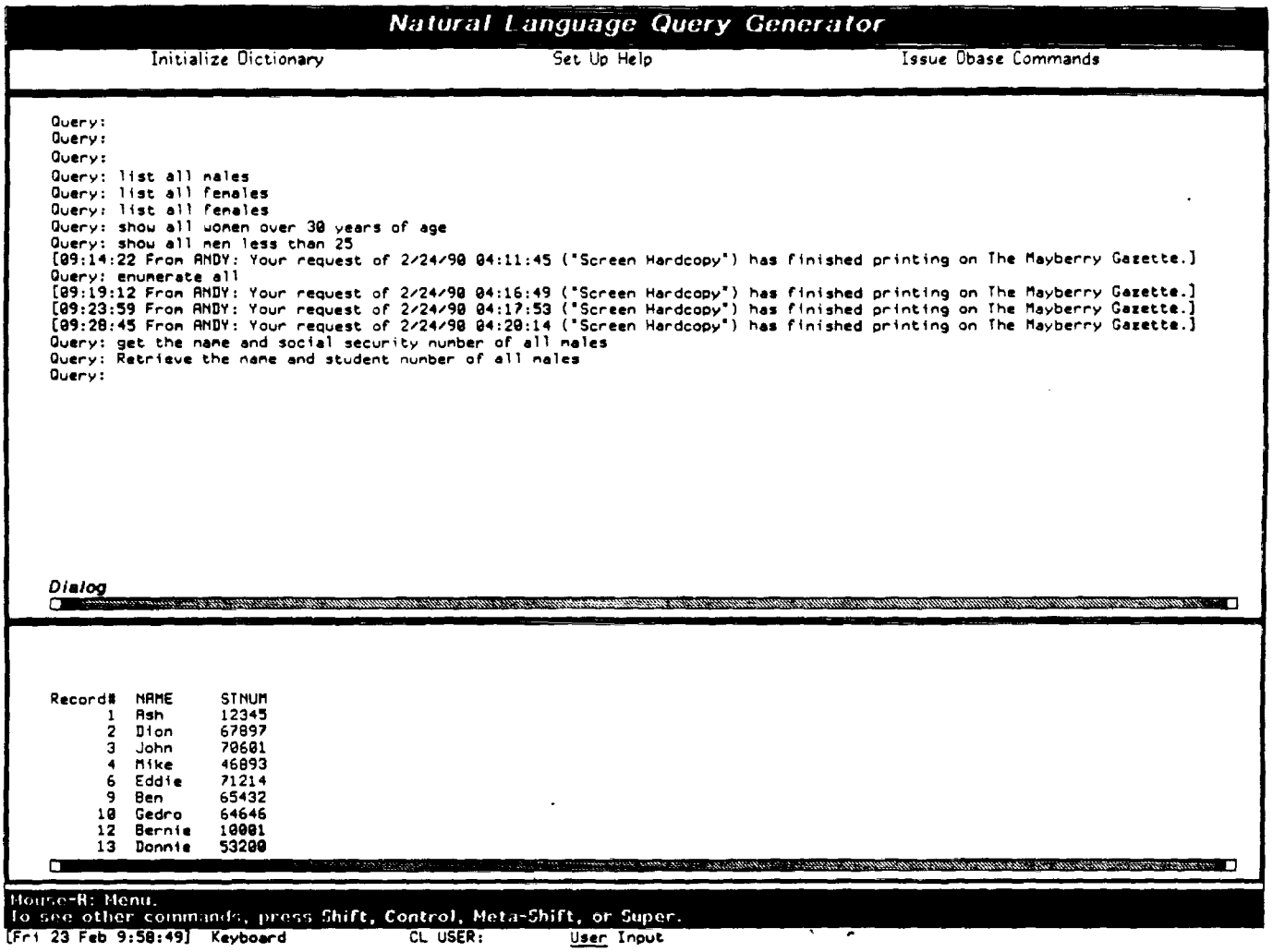


Figure 6

7. To demonstrate the capability of the NLQG to handle different words of the English Language, the following example is used. Instead of "Females" the user uses the word "Women". The NLQG recognizes that women and females mean the same thing and retrieves the name and student number of all women. It is displayed in the following figure:

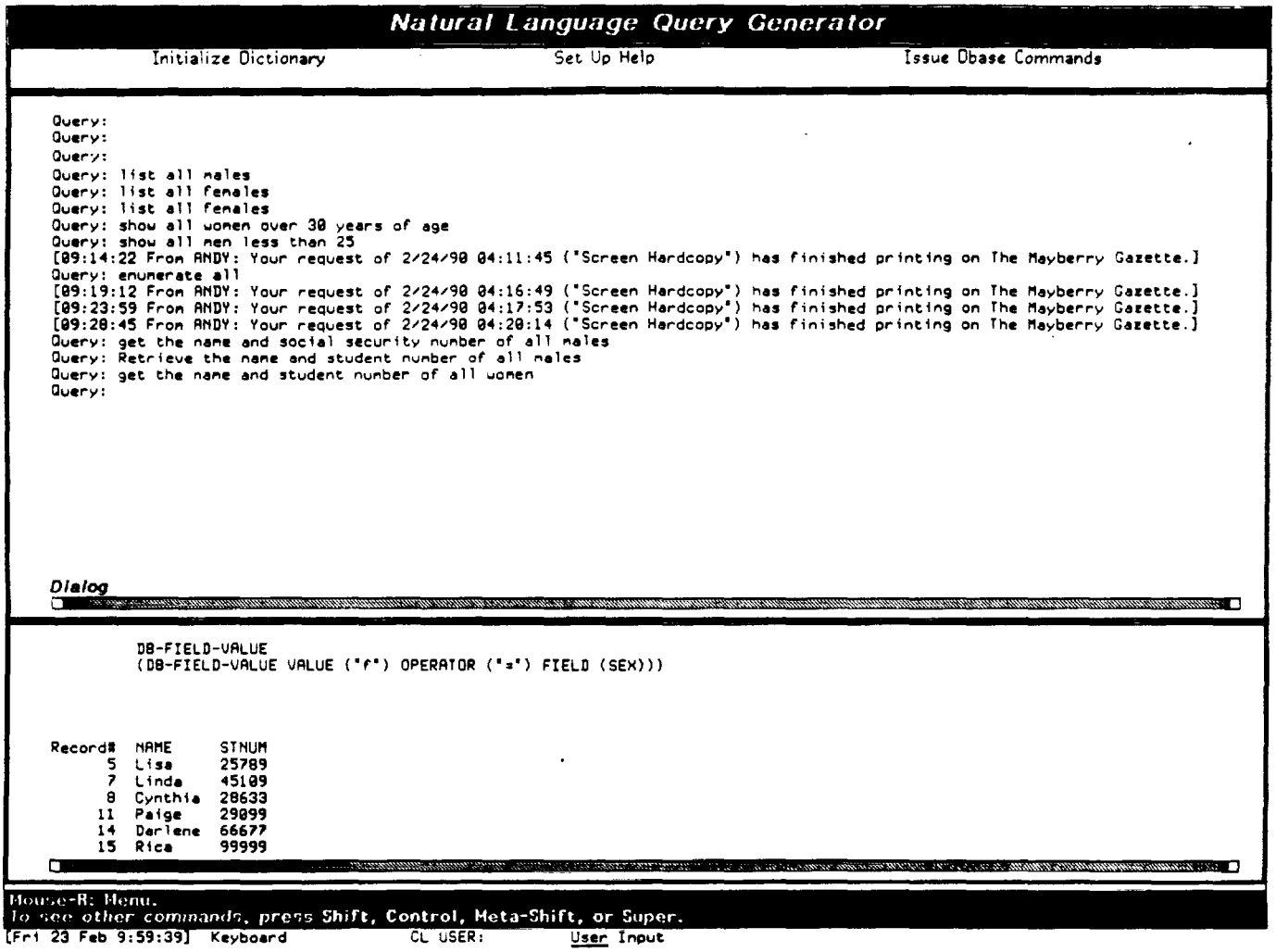


Figure 7

8. In the following example the user uses the word find instead of retrieve or get to display the name and sex of all members of the database. Instead of saying male or female, he/she just types in all and the NLQG retrieves the name and sex of all the members. The following figure illustrates the query and the output.

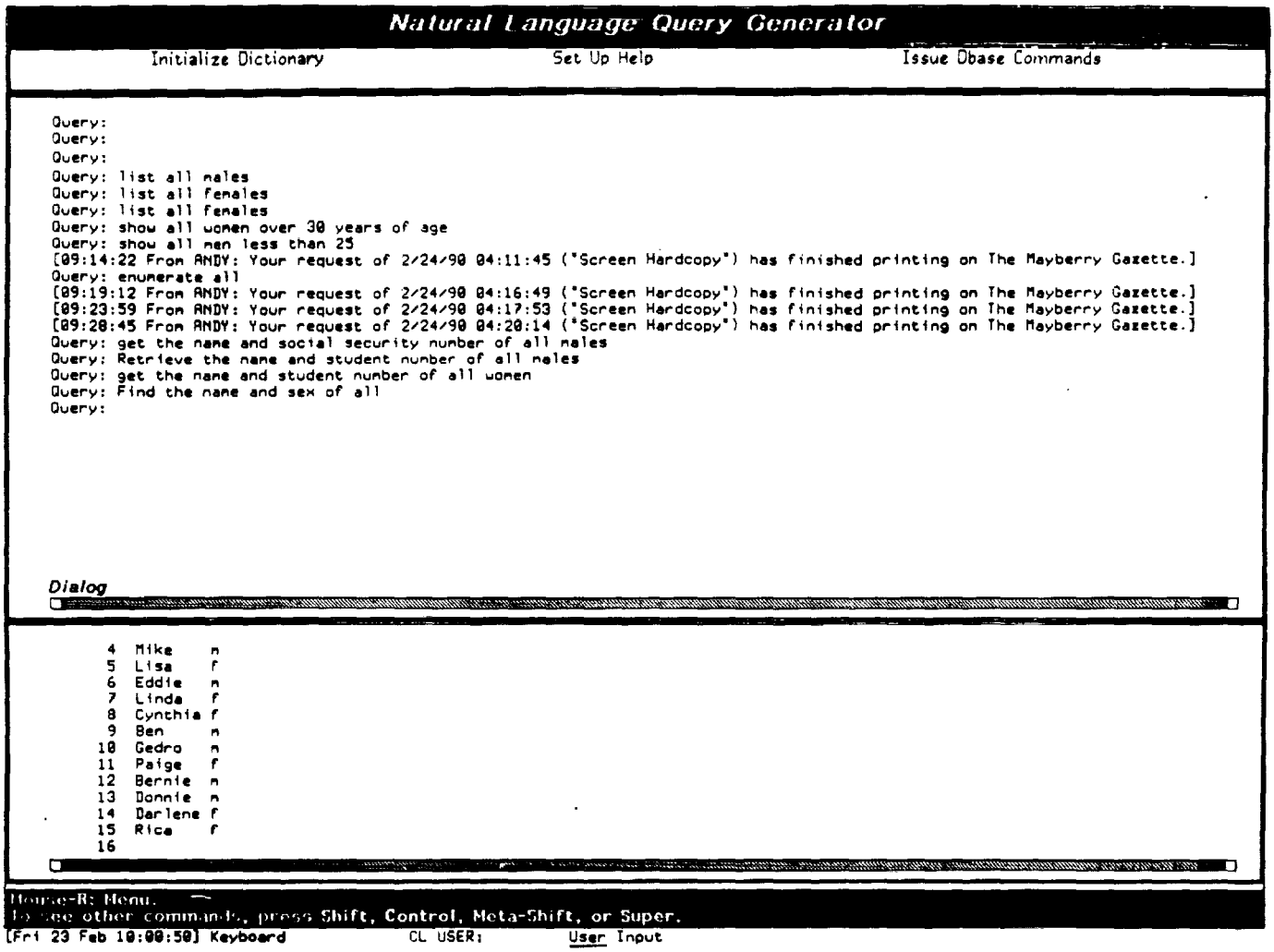


Figure 8

9. This example the user uses the word find to retrieve the name and sex of all males, but uses the word men instead. In addition to this, he also types in the word "please." The NLQG ignores the "please" in that it adds no pertainent new information to the query. The query and output are shown in the following figure.

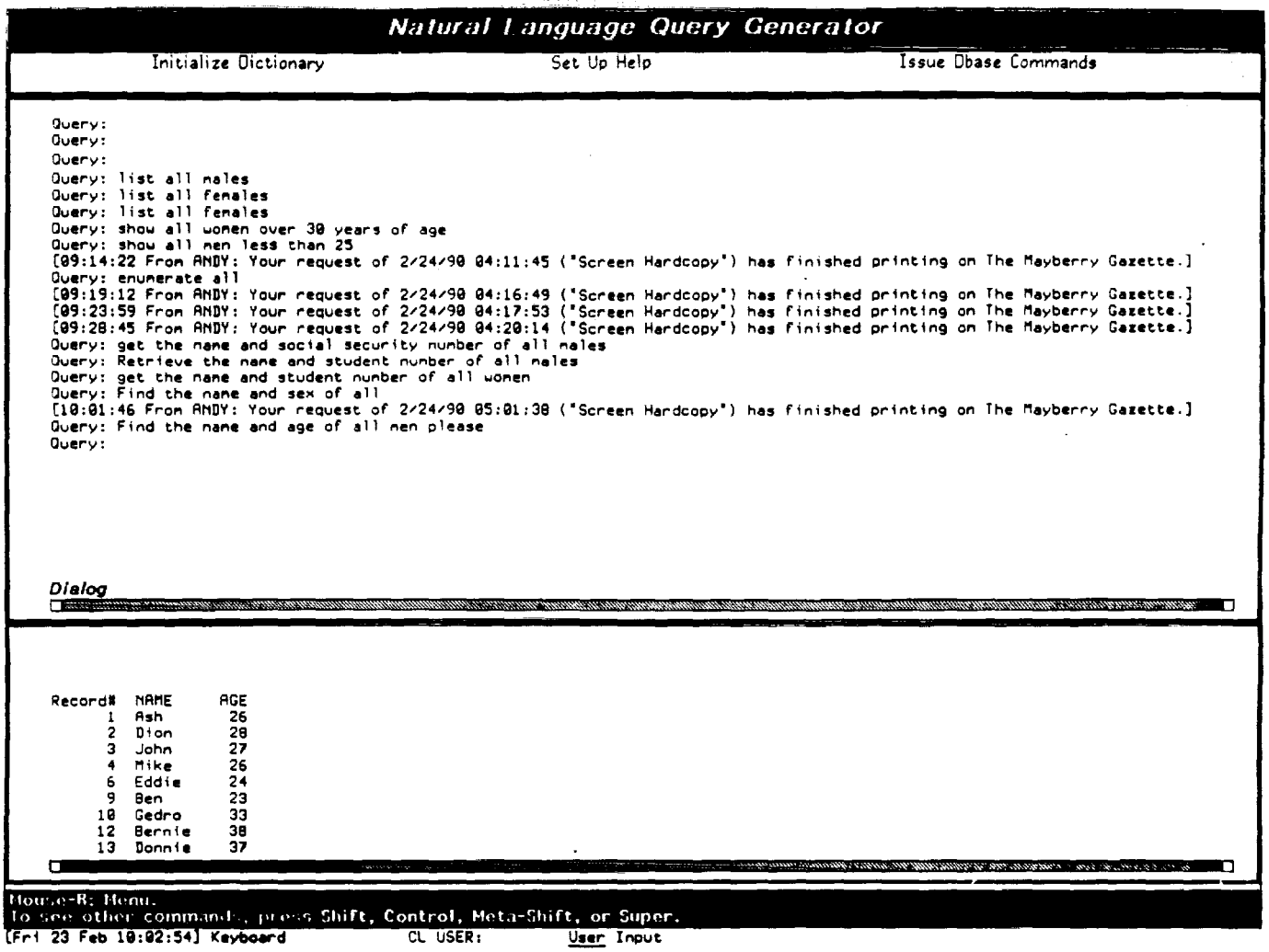


Figure 9

10. In this example the user asks the query in the form of a question. The NLQG retrieves and displays the names of all females. The query and its output are shown in Figure 10.

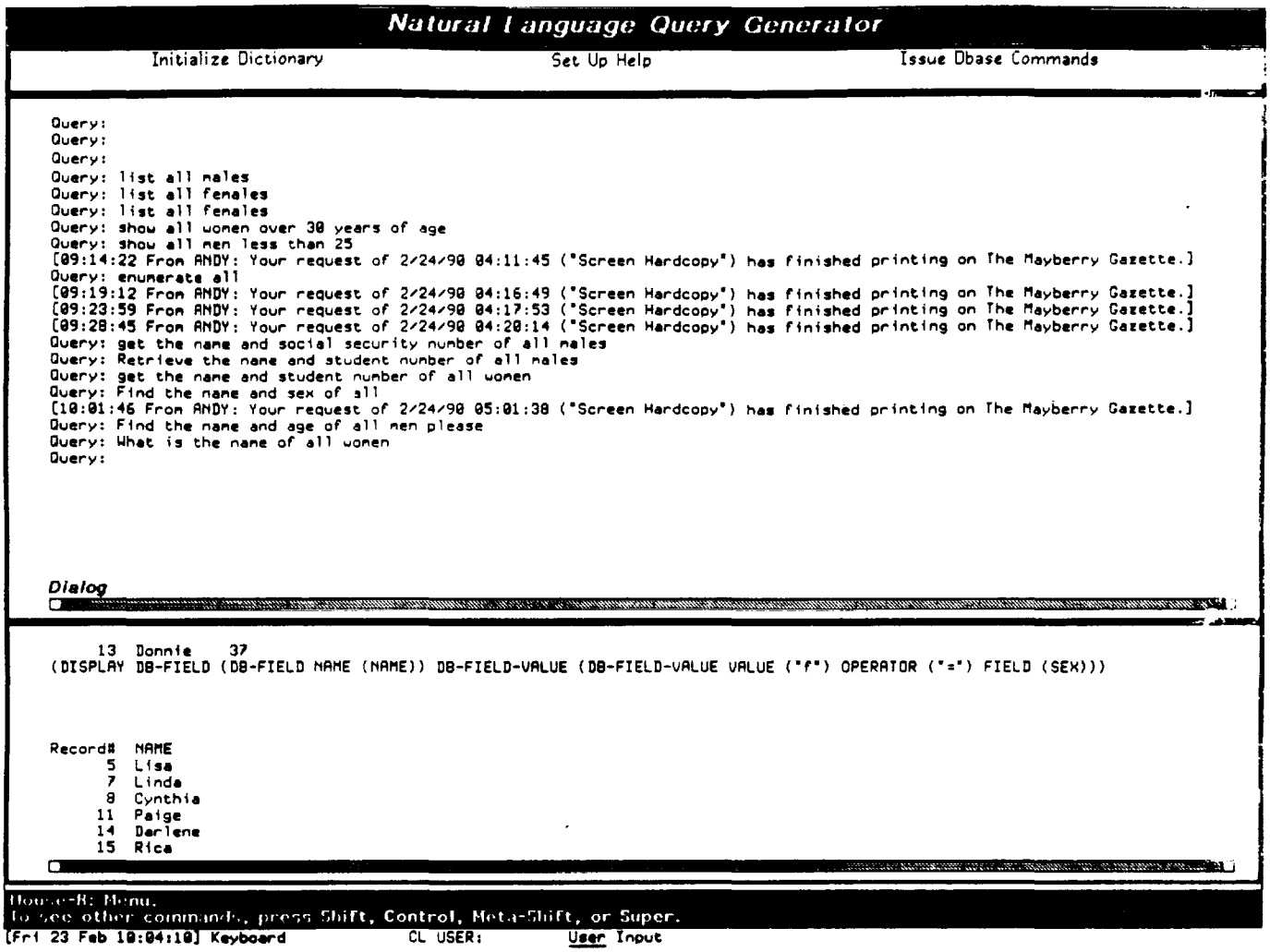


Figure 10

1.4 Task Results

The specifics stated in the Task Statement were successfully completed. The interface was developed and operates on the Symbolic's Lisp machine. The database can be queried from the Symbolic's and the data is returned to the Symbolic's. The interface allows users to query the database in their natural language, if it's English. The interface understands a limited subset of English.

However, novice users can use the interface to query the database, but they still must know some things about the database. For example, they must know the field names used in the construction of the database. Using the above example, the user would have to know that the database contained information about males and females. They would not have to know the exact field name. This is provided for in the dictionary; i.e., other words used to describe the same concept are identified and linked to the appropriate definition. An example of this is using men for male or women for female.

The solution to this type of problem is to develop a generic interface system. However, it is difficult to develop a such a system; i.e., one that will allow the user to simply ask what databases the system knows about and to use any terms to query the system. In order to develop a system like this more time and money needs to be allocated. Another problem associated with the generic system is that the interface has to know about each database and the terms it uses to describe the data. These terms have to be defined in the dictionary. This makes the interface database dependent. In order to make the interface work with another database, these termshave to be redone. Also, if the database changes a similar process must be accomplished.

This approach holds much promise of making database use by novice users simpler. The generic system is not an absurdity. If it is approached

correctly, parts of this concept could prove beneficial to users. The ability to explain what databases it knows about is feasible at present, as well as, being able to develop an interface that will allow a user to teach it about knew databases so that its capabilities can increase. These extensions to the present research would simply require time for development.

Appendix A

Listing of Words Used by the NLI in Lisp Format

```
;;; -*- Syntax: Common-Lisp; Package: COMMON-LISP-USER; Base: 10; Mode: LISP -*-

(learn-words

  '((john      def    (human name   (john)
                              gender (male))
               denons (save-character))

    (pick      denons ((pick-up?)(decide?)(determine-voice))
               m1     (grasp actor  h <==(exp-wrt-voice 'human 'before)
                             object x <==(exp-wrt-voice 'phys-obj 'after)
                             instr  (move actor  h
                                          object (fingers)
                                          to     x))
               m2     (mbuild actor * <==(exp-wrt-voice 'human 'before)
                              nobj  (poss actor  * <==(exp-wrt-voice 'human 'before)
                                          object * <==(exp-wrt-voice '(human phys-obj) 'after))))

    (up        denons (ignor))

    (the       denons (ignor))

    (ball      def    (phys-obj class (game-obj)
                                name  (ball))
               denons (save-object))

    (and def (conjunction conjunct1 * <==(exp-wrt-voice '(db-field phys-obj human) 'before)
                          conjunct2 * <== (exp-wrt-voice '(db-field phys-obj human) 'after))
         denons (determine-voice))

    (dropped   def    (ptrans actor  *   <==(exp-wrt-voice 'human 'before)
                              object thg <==(exp-wrt-voice 'phys-obj 'after)
                              to     *   <==(prep '(in into on) '(human phys-obj) 'after)
                              instr  (propel actor  (gravity)
                                             object thg)))
    (it        def    (pronoun)
               denons (ignor))

    (in        def    (prep is (in))
               denons (ins-aft '(phys-obj setting) 'prepobj))

    (box       def (phys-obj class (container)
                             name  (box)))

    (pcb       def    (process-object name (printed-circuit-board))
               denons ((save-object)(how-many 'quantity 'suffix0 's)))

    (enter     def    (ptrans actor  nil
                              object * <==(exp-wrt-voice '(process-object pronoun) 'before)
                              to     * <==(exp-wrt-voice '(complex process-actor pronoun) 'after))
               denons ((get-sentence-number)(determine-voice)))

    (exit      def    (ptrans actor  nil
                              object * <==(exp-wrt-voice '(process-object pronoun) 'before)
                              from   * <==(exp-wrt-voice '(complex process-actor pronoun) 'after))
               denons ((get-sentence-number)(determine-voice)))

    (process   def    (do actor  * <==(exp-wrt-voice '(process-actor complex pronoun) 'before)
                          object * <==(exp-wrt-voice '(process-object pronoun) 'after))
               denons ((get-sentence-number)(determine-voice)))

    (proceed   def    (ptrans actor  nil
                              object * <==(exp-wrt-voice '(process-object pronoun) 'before)
                              to     * <==(prep '(to) '(complex process-actor pronoun) 'after))
               denons ((get-sentence-number)(determine-voice)))


    (go    def    (ptrans actor  nil
                          object * <==(exp-wrt-voice '(process-object pronoun) 'before)
                          to     * <==(prep '(to) '(complex process-actor pronoun) 'after))
           denons ((get-sentence-number)(determine-voice)))
```

```
(arrive    def    (ptrans actor  nil
                          object * <==(exp-urt-voice '(process-object pronoun) 'before)
                          to     * <==(prep '(at) '(complex process-actor pronoun) 'after))
           demons ((get-sentence-number)(determine-voice)))

(to        def    (prep is (to))
           demons (ins-aft '(process-actor pronoun) 'prepobj))

(at        def    (prep is (at))
           demons (ins-aft '(complex process-actor pronoun) 'prepobj))

(second    def    (time name       (second)
                        base-units (1))
           demons ((attach-time '(dist-type) 'before)(how-many 'quantity 'suffix0 's)))

(minute    def    (time name (minute)
                        base-units (60))
           demons ((attach-time '(dist-type) 'before)(how-many 'quantity 'suffix0 's)))

(hour      def    (time name (hour)
                        base-units (3600))
           demons ((attach-time '(dist-type) 'before)(how-many 'quantity 'suffix0 's)))

(for       def    (prep is (for))
           demons (ignor))

(a         demons (ignor))

(me        demons (ignor))

(then      demons (ignor))

(where     demons (ignor))

(next      demons (ignor))

(is        def    (be-verb name (is))
           demons (ignor))

(are       def    (be-verb name (are))
           demons (ignor))

(by        def    (prep is (by))
           demons (ignor))

(of        def    (prep is (of))
           demons (ignor))

(around    def    (prep is (around))
           demons (ignor))

(near      def    (prep is (near))
           demons (ignor))

(with      def    (prep is (with))
           demons (ignor))

(aic       def     (complex name (automatic-insertion))
           demons  (save-complex))

(mi        def     (complex name (manual-insertion))
           demons  (save-complex))

(mic       def     (complex name (manual-insertion))
           demons  (save-complex))

(testing   def     (complex name (test-and-assembly))
           demons  (save-complex))

(dip       def     (process-actor class (station)
                                  name (dip-machine))
           demons  ((save-actor)(how-many 'quantity 'suffix0 's)))
```

```
(vcd        def     (process-actor class (station)
                                 name (vcd-machine))
            demons  ((save-actor)(how-many 'quantity 'suffix0 's)))

(tdk        def     (process-actor class (station)
                                 name (tdk-machine))
            demons  ((save-actor)(how-many 'quantity 'suffix0 's)))

(rli        def     (process-actor class (station)
                                 name (tdk-machine))
            demons  ((save-actor)(how-many 'quantity 'suffix0 's)))

(bpi        def     (process-actor class (station)
                                 name (berg-pin-machine))
            demons  ((save-actor)(how-many 'quantity 'suffix0 's)))

(swedge     def     (process-actor class (station)
                                 name (swedge-nut-machine))
            demons  ((save-actor)(how-many 'quantity 'suffix0 's)))

(ate        def     (process-actor class (station)
                                 name  (automatic-test))
            demons  (save-actor))

(qc         def     (process-actor class (station)
                                 name  (quality-control))
            demons  (save-actor))

(assembly   def     (process-actor class (station)
                                 name  (mechanical-assembly))
            demons  (save-actor))

(shipping   def     (process-actor class (station)
                                 name  (shipping))
            demons  (save-actor))

(storage    def     (process-actor class (station)
                                 name  (shipping))
            demons  (save-actor))

(mean       def     (statistic name (mean)
                               measure * <==(find-stat-value)))

(sd         def     (statistic name (standard-deviation)
                               measure * <==(find-stat-value)))

(swedge     def     (process-actor class (station)
                                 name (swedge-nut-machine))
            demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

(poisson    def     (dist-type name (poisson)
                               mit * <==(exp-statistic '(mean-interarrival-time) 'after))
            demons  (ins-bef '(ptrans do) 'dist))

(normal     def     (dist-type name (normal)
                               mean * <==(exp-statistic '(mean) 'after)
                               sd * <==(exp-statistic '(standard-deviation) 'after))
            demons  (ins-bef '(ptrans do) 'dist))

(uniform    def     (dist-type name (uniform)
                               min * <==(exp-statistic '(min) 'after)
                               max * <==(exp-statistic '(max) 'after))
            demons  (ins-bef '(ptrans do) 'dist))

(min        def     (statistic name (min)
                               measure * <==(find-stat-value)))

(minimum    def     (statistic name (min)
                               measure * <==(find-stat-value)))

(max        def     (statistic name (max)
                               measure * <==(find-stat-value)))
```

```
(maximum   def     (statistic name (max)
                              measure * <==(find-stat-value)))

(mit       def     (statistic name (mean-interarrival-time)
                              measure * <==(find-stat-value)))


(mike      def      (human name (mike)
                           gender (male))
           demons (save-character))


(ate       def    (ptrans actor  nil
                          object * <==(exp-wrt-voice '(process-object noun) 'before)
                          to     * <==(prep '(a an the) '(complex process-actor noun) 'after))
           demons ((get-sentence-number)(determine-voice)))

(an         def    (prep is (an))
            demons (ignor))

(apple      def     (food type (apple)))


(a         def   (prep is (a))
           demons (ignor))

(wears     def    (ptrans actor * <== (exp-wrt-voice 'human 'before)
                          object thg <==(exp-wrt-voice 'garment after)
                          to     * <==(prep '(a the) '(garment) 'after)
                          demons ((get-sentence-number)(determine-voice))))


(shirt     def    (garment type (shirt)))


(retrieve  def   (ptrans actor nil
                          object * <==(exp-wrt-voice '(process-object pronoun) 'before)
                          to     * <==(prep '(from) '(process actor class) 'after))
            demons ((get-sentence-number)(determine-voice)))

(get       def   (ptrans actor nil
                          object * <==(exp-wrt-voice '(process-object pronoun) 'before)
                          to     * <==(prep '(from) '(complex process-actor pronoun) 'after))
           demons ((get-sentence-number)(determine-voice)))


(shot       m1  (propel object (bullets))
            def    (ptrans actor nil
                           object * <==(exp-wrt-voice '(process-object pronoun) 'before)
                           to     * <==(prep '(the) '(complex process-actor noun) 'after))
            demons ((get-sentence-number)(determine-voice))
            m2  ($ take-picture))


(from      demons (ignor))


(insert  def   (ptrans actor nil
                       object * <==(exp-wrt-voice '(process-object pronoun) 'before)
                       to     * <==(prep '(in) '(process actor class) 'after))
         demons ((get-sentence-number)(determine-voice)))


(delete  def   (ptrans actor nil
                       object * <==(exp-wrt-voice '(process-object pronoun) 'before)
                       to     * <==(prep '(from) '(complex process-actor pronoun) 'after))
         demons ((get-sentence-number)(determine-voice)))

(modify  def (ptrans actor nil
                     object * <==(exp-wrt-voice '(process-object pronoun) 'before)
```

```
                    (setq 'actor ( * 1  actor)
                          to     * <==(prep '(from) '(complex process-actor pronoun) 'after))))

(print  def    (ptrans actor (display)
                       object * <==(exp-wrt-voice '(db-field) 'after)
                       objects * <==(exp-wrt-voice '(conjunction) 'after))
       denons ((get-sentence-number) (determine-voice)))

(fetch  def    (ptrans actor (display)
                       object * <==(exp-wrt-voice '(db-field) 'after)
                       objects * <==(exp-wrt-voice '(conjunction) 'after))
       denons ((get-sentence-number) (determine-voice)))

(retrieve  def    (ptrans actor (display)
                          object * <==(exp-wrt-voice '(db-field) 'after)
                          objects * <==(exp-wrt-voice '(conjunction) 'after))
          denons ((get-sentence-number) (determine-voice)))

(get  def    (ptrans actor (display)
                     object * <==(exp-wrt-voice '(db-field) 'after)
                     objects * <==(exp-wrt-voice '(conjunction) 'after))
     denons ((get-sentence-number) (determine-voice)))

(show  def    (ptrans actor (display)
                       object * <==(exp-wrt-voice '(db-field) 'after)
                       objects * <==(exp-wrt-voice '(conjunction) 'after))
      denons ((get-sentence-number) (determine-voice)))

(select  def    (ptrans actor (display)
                        object * <==(exp-wrt-voice '(db-field) 'after)
                        objects * <==(exp-wrt-voice '(conjunction) 'after))
        denons ((get-sentence-number) (determine-voice)))

(list  def    (ptrans actor (display)
                       object * <==(exp-wrt-voice '(db-field) 'after)
                       objects * <==(exp-wrt-voice '(conjunction) 'after))
      denons ((get-sentence-number) (determine-voice)))

(enumerate  def    (ptrans actor (display)
                           object * <==(exp-wrt-voice '(db-field) 'after)
                           objects * <==(exp-wrt-voice '(conjunction) 'after))
           denons ((get-sentence-number) (determine-voice)))

(display  def    (ptrans actor (display)
                         object * <==(exp-wrt-voice '(db-field) 'after)
                         objects * <==(exp-wrt-voice '(conjunction) 'after))
          denons ((get-sentence-number) (determine-voice)))

(give  def    (ptrans actor (display)
                       object * <==(exp-wrt-voice '(db-field) 'after)
                       objects * <==(exp-wrt-voice '(conjunction) 'after))
      denons ((get-sentence-number) (determine-voice)))

(all  denons (*ignor*))

(age  def    (db-field name (age))
     denons (save-object))
(sex  def    (db-field name (sex))
     denons (save-object))
(name  def    (db-field name (name))
      denons (save-object))
(people  def    (db-field name (name))
        denons (save-object))
(person  def    (db-field name (name))
        denons (save-object))

(male  def   (db-field-value value ("m")
                             field (sex))
      denons (ins-bef '(ptrans) 'db-field-value))

(female  def   (db-field-value value ("f")
                               field (sex))
        denons (ins-bef '(ptrans) 'db-field-value))
```

```
(man  def   (db-field-value value ("m")
                            field (sex))
      demons (ins-bef '(ptrans) 'db-field-value))
(woman  def   (db-field-value value ("f")
                              field (sex))
        demons (ins-bef '(ptrans) 'db-field-value))

(table        def    (process-object name (row)))))
```

Appendix B

Listing of Expressions Used by the NLI in Lisp Format

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: COMMON-LISP-USER; Base: 10 -*-

(learn-expressions

  '(((automatic insertion center)
     def     (complex name (automatic-insertion))
     denons  (save-complex))

    ((ai work center)
     def     (complex name (automatic-insertion))
     denons  (save-complex))

    ((automatic insertion)
     def     (complex name (automatic-insertion))
     denons  (save-complex))

    ((automatic insertion work center)
     def     (complex name (automatic-insertion))
     denons  (save-complex))

    ((ai center)
     def     (complex name (automatic-insertion))
     denons  (save-complex))

    ((manual insertion center)
     def     (complex name (manual-insertion))
     denons  (save-complex))

    ((manual insertion)
     def     (complex name (manual-insertion))
     denons  (save-complex))

    ((manual insertion work center)
     def     (complex name (manual-insertion))
     denons  (save-complex))

    ((manual load)
     def     (complex name (manual-insertion))
     denons  (save-complex))

    ((manual load center)
     def     (complex name (manual-insertion))
     denons  (save-complex))

    ((manual load work center)
     def     (complex name (manual-insertion))
     denons  (save-complex))

    ((test and assembly)
     def     (complex name (test-and-assembly))
     denons  (save-complex))

    ((test and assembly center)
     def     (complex name (test-and-assembly))
     denons  (save-complex))

    ((test and assembly work center)
     def     (complex name (test-and-assembly))
     denons  (save-complex))

    ((testing work center)
     def     (complex name (test-and-assembly))
     denons  (save-complex))

    ((testing center)
     def     (complex name (test-and-assembly))
     denons  (save-complex))

    ((testing and assembly work center)
     def     (complex name (test-and-assembly))
     denons  (save-complex))
```

```
((testing and assembly center)
 def     (complex name (test-and-assembly))
 demons  (save-complex))

((testing and assembly)
 def     (complex name (test-and-assembly))
 demons  (save-complex))

((t & a)
 def     (complex name (test-and-assembly))
 demons  (save-complex))

((t & a work center)
 def     (complex name (test-and-assembly))
 demons  (save-complex))

((t & a center)
 def     (complex name (test-and-assembly))
 demons  (save-complex))

((finished goods)
 def     (complex name (finished-goods))
 demons  (save-complex))

((dip machine)
 def     (process-actor class (station)
                        name (dip-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((dual inline package insertion machine)
 def     (process-actor class (station)
                        name (dip-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix40 's)))

((dual in-line package insertion machine)
 def     (process-actor class (station)
                        name (dip-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix40 's)))

((dual inline package insertion)
 def     (process-actor class (station)
                        name (dip-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix30 's)))

((dual in-line package insertion)
 def     (process-actor class (station)
                        name (dip-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix30 's)))

((dip insertion)
 def     (process-actor class (station)
                        name (dip-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((dip insertion machine)
 def     (process-actor class (station)
                        name (dip-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((masking machine)
 def     (process-actor class (station)
                        name (masking-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((masking)
 def     (process-actor class (station)
                        name (masking-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((vcd machine)
 def     (process-actor class (station)
                        name (vcd-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))
```

```
((vcd insertion machine)
 def     (process-actor class (station)
                        name (vcd-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((vcd insertion)
 def     (process-actor class (station)
                        name (vcd-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((variable center distance insertion)
 def     (process-actor class (station)
                        name (vcd-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix30 's)))

((variable center distance insertion machine)
 def     (process-actor class (station)
                        name (vcd-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix40 's)))

((tdk machine)
 def     (process-actor class (station)
                        name (tdk-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((radial lead insertion machine)
 def     (process-actor class (station)
                        name (tdk-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix30 's)))

((radial lead insertion)
 def     (process-actor class (station)
                        name (tdk-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((rli machine)
 def     (process-actor class (station)
                        name (tdk-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((rl insertion machine)
 def     (process-actor class (station)
                        name (tdk-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((radial lead machine)
 def     (process-actor class (station)
                        name (tdk-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((radial lead)
 def     (process-actor class (station)
                        name (tdk-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((berg pin machine)
 def     (process-actor class (station)
                        name (berg-pin-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((berg pin)
 def     (process-actor class (station)
                        name (berg-pin-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((berg machine)
 def     (process-actor class (station)
                        name (berg-pin-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((berg pin insertion machine)
 def     (process-actor class (station)
                        name (berg-pin-machine))
```

```
 denons  ((save-actor)(how-many 'quantity 'suffix30 's)))

((berg pin insertion)
 def     (process-actor class (station)
                        name (berg-pin-machine))
 denons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((bpi machine)
 def     (process-actor class (station)
                        name (berg-pin-machine))
 denons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((bp machine)
 def     (process-actor class (station)
                        name (berg-pin-machine))
 denons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((swedge nut machine)
 def     (process-actor class (station)
                        name (swedge-nut-machine))
 denons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((swedge nut)
 def     (process-actor class (station)
                        name (swedge-nut-machine))
 denons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((swedge machine)
 def     (process-actor class (station)
                        name (swedge-nut-machine))
 denons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((spanish terminal insertion machine)
 def     (process-actor class (station)
                        name (swedge-nut-machine))
 denons  ((save-actor)(how-many 'quantity 'suffix30 's)))

((spanish terminal insertion)
 def     (process-actor class (station)
                        name (swedge-nut-machine))
 denons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((spanish terminal)
 def     (process-actor class (station)
                        name (swedge-nut-machine))
 denons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((spanish terminal machine)
 def     (process-actor class (station)
                        name (swedge-nut-machine))
 denons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((component preparation)
 def     (process-actor class (station)
                        name (component-preparation))
 denons  (save-actor))

((component prep)
 def     (process-actor class (station)
                        name (component-preparation))
 denons  (save-actor))

((hand load)
 def     (process-actor class (station)
                        name  (manual-load))
 denons  (save-actor))

((manual load)
 def     (process-actor class (station)
                        name  (manual-load))
 denons  (save-actor))

((wave solder machine)
```

```
 def     (process-actor class (station)
                        name  (wave-solder-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((wave solder)
 def     (process-actor class (station)
                        name  (wave-solder-machine))
 demons  (save-actor))

((aqua clean machine)
 def     (process-actor class (station)
                        name  (aqua-clean-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix20 's)))

((aqua clean)
 def     (process-actor class (station)
                        name  (aqua-clean-machine))
 demons  (save-actor))

((aqua bath)
 def     (process-actor class (station)
                        name  (aqua-clean-machine))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((secondary operations)
 def     (process-actor class (station)
                        name  (secondary-operations))
 demons  (save-actor))

((secondary ops)
 def     (process-actor class (station)
                        name  (secondary-operations))
 demons  (save-actor))

((quality control station)
 def     (process-actor class (station)
                        name  (quality-control))
 demons  (save-actor))

((qc station)
 def     (process-actor class (station)
                        name  (quality-control))
 demons  (save-actor))

((qc point)
 def     (process-actor class (station)
                        name  (quality-control))
 demons  (save-actor))

((repair station)
 def     (process-actor class (station)
                        name  (repair-station))
 demons  (save-actor))

((fault finder)
 def     (process-actor class (station)
                        name  (fault-finder))
 demons  ((save-actor)(how-many 'quantity 'suffix10 's)))

((burn in)
 def     (process-actor class (station)
                        name  (burn-in))
 demons  (save-actor))

((automatic test)
 def     (process-actor class (station)
                        name  (automatic-test))
 demons  (save-actor))

((first functional test)
 def     (process-actor class (station)
                        name  (automatic-test))
 demons  (save-actor))
```

```
((mechanical assembly)
 def     (process-actor class (station)
                        name  (mechanical-assembly))
 demons  (save-actor))

((final functional test)
 def     (process-actor class (station)
                        name  (final-functional-test))
 demons  (save-actor))

((final inspection)
 def     (process-actor class (station)
                        name  (final-inspection))
 demons  (save-actor))

((button up)
 def     (process-actor class (station)
                        name  (button-up))
 demons  (save-actor))

((according to)
 demons  (ignor))

((poisson process)
 def     (dist-type name (poisson)
                    mit * <==(exp-statistic '(mean-interarrival-time) 'after))
 demons  (ins-bef '(ptrans do) 'dist))

((poisson distribution)
 def     (dist-type name (poisson)
                    mit * <==(exp-statistic '(mean-interarrival-time) 'after))
 demons  (ins-bef '(ptrans do) 'dist))

((poisson model)
 def     (dist-type name (poisson)
                    mit * <==(exp-statistic '(mean-interarrival-time) 'after))
 demons  (ins-bef '(ptrans do) 'dist))

((normal process)
 def     (dist-type name (normal)
                    mean * <==(exp-statistic '(mean) 'after)
                    sd * <==(exp-statistic '(standard-deviation) 'after))
 demons  (ins-bef '(ptrans do) 'dist))

((normal distribution)
 def     (dist-type name (normal)
                    mean * <==(exp-statistic '(mean) 'after)
                    sd * <==(exp-statistic '(standard-deviation) 'after))
 demons  (ins-bef '(ptrans do) 'dist))

((normal model)
 def     (dist-type name (normal)
                    mean * <==(exp-statistic '(mean) 'after)
                    sd * <==(exp-statistic '(standard-deviation) 'after))
 demons  (ins-bef '(ptrans do) 'dist))

((uniform distribution)
 def     (dist-type name (uniform)
                    min * <==(exp-statistic '(min) 'after)
                    max * <==(exp-statistic '(max) 'after))
 demons  (ins-bef '(ptrans do) 'dist))

((uniform process)
 def     (dist-type name (uniform)
                    min * <==(exp-statistic '(min) 'after)
                    max * <==(exp-statistic '(max) 'after))
 demons  (ins-bef '(ptrans do) 'dist))

((uniform model)
 def     (dist-type name (uniform)
                    min * <==(exp-statistic '(min) 'after)
                    max * <==(exp-statistic '(max) 'after))
 demons  (ins-bef '(ptrans do) 'dist))
```

```
((mininum value)
 def     (statistic name (min)
                    measure * <==(find-stat-value)))

((maximum value)
 def     (statistic name (max)
                    measure * <==(find-stat-value)))

((printed circuit board)
 def     (process-object name (printed-circuit-board))
 demons  ((save-object)(how-many 'quantity 'suffix20 's)))

((standard deviation)
 def     (statistic name    (standard-deviation)
                    measure * <==(find-stat-value))
 demons  (ignor))

((std dev)
 def     (statistic name    (standard-deviation)
                    measure * <==(find-stat-value))
 demons  (ignor))

((mean interarrival time)
 def     (statistic name (mean-interarrival-time))
 measure * <==(find-stat-value))



((memory)
 def     (process-actor class (station)
                        name (memory))
 demons  ((save-actor)(how-many 'quantity 'suffix1 's)))


((Data-base)
 def (process-actor class (station)
                    name (data-base))
 demons ((save-actor)(how-many 'quantity 'suffix1 's)))

((student number)  def    (db-field name (stnum))
 demons (save-object))

((social security number)  def    (db-field name (ssnum))
 demons (save-object))))
```